intel
AR-236

LET OPERATING SYSTEMS AID IN COMPONENT DESIGNS

The iRMX 86 operating system processor package offers hardware designers a set of thoroughly tested software primitives upon which to build present and future custom hardware designs

by George Heider

Component users build application systems by integrating standard and custom hardware, software, and packaging. Microprocessors and other very large scale integration components are replacing much custom hardware with larger, more powerful standard hardware modules. Microprocessors lead to powerful systems, but they often require complex system management software. While this complex software often comprises one third or less of the final system software, it may require two thirds or more of the storage development effort. Worse, bugs in system management software sometimes do not show up until late in development or after the product is at the customer's site.

One solution to this problem is to employ standard management software such as operating systems. More complex, multifunction applications in a realtime environment benefit greatly from operating systems. Examples of these applications include file subsystems, public automatic branch exchange (PABX) systems, and transaction processing systems. But implementing operating system functions in a component design requires new software tools, education, and expertise. Also, these functions are often specific to the particular design, so tools and expertise developed for one application are not suitable for subsequent designs.

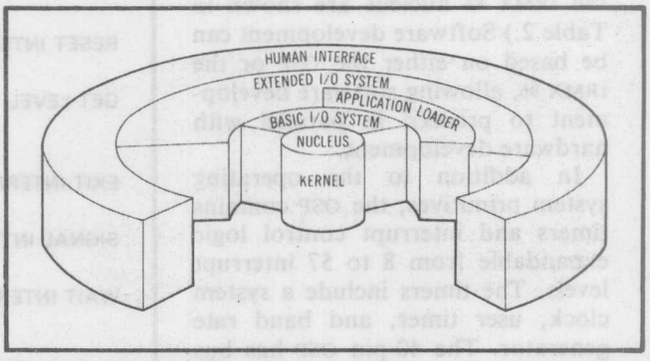


Fig 1 iRMX operating system architecture. Kernel consists of primitives also implemented in hardware in iAPX 86/30 and iAPX 88/30 OSP.

These problems are directly addressed by the Intel iAPX 86/30 and iAPX 88/30 operating system processors (OSP) and the iRMX* 86 operating system. The iRMX 86 is a full-featured, realtime multitasking operating system for iAPX 86 or iAPX 88 based systems. The Intel OSP implements the iRMX 86 kernel functions in hardware consisting of an iAPX 86 or iAPX 88 central processor coupled with an operating system firmware (OSF) component, the Intel 80130. The OSF extends the base iAPX 86 and iAPX 88 architecture by adding 35 operating system primitive instructions to the base iAPX 86 or iAPX 88 instruction set; systems can be built directly on the OSP. System implementation time is thus decreased by having fully debugged operating system functions in hardware.

George Heider is a senior applications engineer at Intel's OEM Microcomputer Systems Div, 5200 NE Elam Young Pkwy, Hillsboro, OR 97123. He works primarily with 16-bit software applications, including the iRMX 86 operating system. Previous experience includes telecommunications systems engineering, microprocessor systems, microprocessor operating system development, and disk storage system software. Mr Heider holds an MS in computer science from the University of California, Santa Barbara and a BSEE from Oregon State University.

Further capabilities can be added by extending the set of OSP primitives or by integrating portions of the iRMX 86 on top of the OSP.

Operating system architecture

The iRMX 86 architecture shown in Fig 1 consists of the nucleus and layers for the basic input/output (I/O) system, extended I/O system, application loader, and human interface. The system also provides a debugger, a terminal handler, a bootstrap loader, and a patch facility.

While the nucleus is the lowest layer of the operating system, fundamental system functions are handled by the nucleus kernel, which is the core of any operating system. The kernel controls memory allocation, allocates processor resources, communicates between processes, and manages interrupts. In the Intel OSP these functions are implemented in hardware. (OSP functions are described in Table 1; additional functions supported by the iRMX 86 nucleus are shown in Table 2.) Software development can be based on either the OSP or the iRMX 86, allowing software development to proceed in parallel with hardware development.

In addition to the operating system primitives, the OSP contains timers and interrupt control logic expandable from 8 to 57 interrupt levels. The timers include a system clock, user timer, and baud rate generator. The 40-pin OSP has bus buffers and demultiplex logic, which allows it to interface directly to the iAPX 86 or iAPX 88 multiplexed bus. The OSP can be located at any 16k-byte address boundary in the 1M-byte system address space. Application interface to OSP stepping and revision levels is independent. A block diagram of the 80130 is shown in Fig 2.

TABLE 1

OSP primitives

Primitive	Description
JOB	
CREATE JOB	Creates a job partition including memory pool, task list, and stack area.
TASK	
CREATE TASK	Creates a task with specified environment and priority. Task is created in ready state. Checks for insufficient memory available within containing job.
DELETE TASK	Deletes a task from system as well as from any queues it is awaiting. Task's state and stack segment are deallocated.
SUSPEND TASK	Suspends a task (changes its status to suspended) or increases task's suspension count by 1. A sleeping task may also be suspended and will awaken suspended unless resumed.
RESUME TASK	Decreases suspension count of a task by 1. If at that point count is reduced to 0, task state is made ready. If it was suspend-asleep, it is put back to sleep.
SLEEP	Puts task in asleep state; up to 10 ms units can be specified.
GET TASK TOKENS	Gives token for a task or task's job partition.
INTERRUPT	
SET PRIORITY	Changes task's priority to value passed in primitive.
SET INTERRUPT	Assigns an interrupt handler to a level. Task that makes this call is made interrupt task for same level, unless call indicates there is no interrupt task.
RESET INTERRUPT	Disables an interrupt level; cancels interrupt handler; deletes interrupt task for level if assigned.
GET LEVEL	Returns number of the interrupt level for highest priority interrupt handler currently in operation (several interrupt handlers can be operating).
EXIT INTERRUPT	Completes interrupt processing and sends end of interrupt signal to hardware.
SIGNAL INTERRUPT	Invokes interrupt task assigned to a level from that level's interrupt handler.
WAIT INTERRUPT	Suspends interrupt task state pending a signal interrupt from an interrupt handler. Used by an interrupt task to signal its readiness to service an interrupt.
ENTER INTERRUPT	Sets data segment base for an interrupt handler.
ENABLE	Enables external interrupt level.
DISABLE	Disables an external interrupt level.
GET EXCEPTION HANDLER	Reads location and exception handling mode of current OSP exception handler for a task.
SET EXCEPTION HANDLER	Establishes location and exception handling mode of current OSP exception handler for task.
SIGNAL EXCEPTION	Notifies current OSP exception handler of exception.

Minimum hardware requirements for the iRMX 86 operating system shown in Fig 3 are 1.8k bytes of random access memory (RAM), about 16k bytes of kernel code memory, and integrated circuits. By comparison, the OSP shown in Fig 4 still requires 1.8k bytes of RAM, but does not require the kernel code, the programmable interrupt controller, or the programmable interrupt timer. These are all replaced by the OSP. Approximately 1k bytes of required system configuration code are not shown in Figs 3 and 4.

Kernel functions

Since it defines system architecture, application requests for system operations like interrupt management and memory allocation must go through the kernel. These requests are made by system calls, or primitives, which are comparable to subroutine calls for system actions. Since the kernel manages much of the system hardware, the application code need not concern itself with many hardware details. This independence is not absolute, however: system hardware or resources not managed by the kernel still require application code.

Basic kernel concepts can be explained using a general purpose system (Fig 5). Input data can be characters, analog signals, or digital signals; processing can be numerical analysis, editing, spectrum analysis, process control algorithms, or virtually any other transformation. Processed data must be sent to an interrupt driven output device—a display, a communications line,

Primitive	Description
SEGMENT	
CREATE SEGMENT	Dynamically allocates area of memory of specified length in 16-byte paragraph units up to 64k-byte maximum (eg, for use as buffer). Returns location token for segment allocated.
DELETE SEGMENT	Deallocates memory segment indicated by parameter token.
ENABLE DELETION	Allows deletion of system data type value indicated by location token.
DISABLE DELETION	Prevents deletion of system data type value indicated by location token.
MAILBOX	
CREATE MAILBOX	Creates a mailbox with specified task queuing discipline. Returns location token.
DELETE MAILBOX	Deletes a mailbox and returns its memory. If tasks are waiting for mailbox, they are awakened (ie, their state is made ready) with appropriate exception condition. If messages are waiting for tasks, they are discarded.
SEND MESSAGE	Sends message segment to mailbox.
RECEIVE MESSAGE	Task is ready to receive message at mailbox. Task is placed on mailbox task queue. Task can wait for response indefinitely, wait (generally 10 ms) units, or not wait. When complete, primitive returns to task the location token of message segment received.
REGION	
CREATE REGION	Creates region data type value, specifying queuing discipline. Returns token for region.
DELETE REGION	Deletes region if the region is not in use.
ACCEPT CONTROL	Gains control of region if region immediately available, but does not wait if not available.
RECEIVE CONTROL	Same primitive as accept control but task that performs it may elect to wait.
SEND CONTROL	Relinquishes region.
OTHER	
SET OS EXTENSION	Links new primitive with kernel.
GET TYPE	Gives system type code of a system data type.

control hardware, or mass storage. In this general purpose system, input, process, and output are the only functions, or tasks, that make up the system.

Buffer management

Assume input data will be placed into 128-byte buffers by the input task. Without help from the operating system, the buffers must be prelocated in RAM. Software is needed to manage the buffers, which must be given to the tasks in the correct sequence and returned for reuse when empty. If the buffers are too small, or if RAM is moved, the software must handle these changes.

If an operating system or OSP is used, the locations and sizes of RAM are made known to the kernel during system initialization. The input task requests each buffer, or memory segment, from the kernel by making the kernel system call "create segment" with 128 bytes. If a larger buffer is needed, the create segment call needs a larger value for the size parameter. When the buffer is full, the input task gives the segment to the process task. When the buffer is no longer needed, it can be returned to the system memory pool by a "delete segment" system call. Because the kernel dynamically manages memory allocation and buffer access, no additional code for these functions is necessary.

Communication and synchronization through mailboxes

The sample system needs a dispatching algorithm to send the segments from task to task. Such an algorithm can be written without an operating system. For example, the input task can fill a buffer and call the process task. When the process task finishes, it can call the output task; the output task can finish with the buffer and return. When control returns to the input task, system processing for that buffer is complete. Another method is to have a polling task occasionally check if buffers are ready to be sent to other tasks. Both methods are inefficient and rigid, requiring that each task finish processing data in each buffer before another task can run.

With an operating system, the buffers can be sent from task to task through "mailboxes"—places where tasks can send or receive data. (See Fig 6.) Task A sends a message (segment) to mailbox 1 and specifies mailbox 2 as a return mailbox. Task A then waits for a return message at mailbox 2. Task B receives the message (segment) from mailbox 1, then sends a return message with status to mailbox 2. Task A receives the return message, which contains task B status, and synchronizes the two tasks.

In general, each task obtains a segment, modifies its contents, sends the segment to the next task, and waits for another segment. The input task first gets a segment using "create segment." When the segment is full, the input task uses the kernel call "send message" to send the segment to mailbox A. The process task uses the "receive message" system call to wait at mailbox A for the segment. The process task receives the segment, processes the data, puts the new data in the segment, and sends the segment to mailbox B. The process task then waits at mailbox A for the next segment from the input task. The output task takes the segment from mailbox B

TABLE 2

Additional primitives supported by the iRMX 86 nucleus

CATALOGING SYSTEM DATA TYPES	
CATALOG OBJECT	Catalogs system data type token under name given by task in job partition directory.
UNCATALOG OBJECT	Removes name and token from job partition directory.
LOOKUP OBJECT	Uses name to find token cataloged in job partition directory.
NEW SYSTEM DATA TYPES	
CREATE EXTENSION	Notifies kernel of new system data type code for new system data type.
DELETE EXTENSION	Removes system data type code and deletes all composite system data types with that system data type code.
CREATE COMPOSITE	Creates new system data type from list of current system data types and system data type code received from create extension.
DELETE COMPOSITE	Deletes new system data type.
INSPECT COMPOSITE	Gives list of system data types that form new system data type.
ALTER COMPOSITE	Changes list of system data types that form new system data type.
SEMAPHORES	
CREATE SEMAPHORE	Creates semaphore system data type.
DELETE SEMAPHORE	Deletes semaphore system data type.
SEND UNITS	Task adds a number of units to semaphore.
RECEIVE UNITS	Task asks for a number of units from semaphore. Task can wait for response indefinately, wait (generally 10 ms), or not wait.
OTHER PRIMITIVES	
GET PRIORITY	Gives priority level of task.
FORCE DELETE	Deletes system data type even if disabled delete has been called for system data type.
GET SIZE	Gives byte size of memory segment.
ADDITIONAL JOB PRIMITIVES	
OFFSPRING	Returns child job partitions created by a task in parent job partition.
GET POOL ATTRIBUTES	Gives memory pool attributes of job partition, including pool minimum, pool maximum, initial size, number of bytes used, and number of bytes available.
SET POOL MINIMUM	Changes pool minimum for job partition.
DELETE JOB	Deletes job partition and returns its memory to parent job partition.

and outputs the data. The output task has two choices: it can either delete the segment, letting the input task create more segments, or it can send the segment to mailbox C. After sending or deleting the segment, the output task waits at mailbox B for the next segment from the process task. If the output task sent the segment to mailbox C, the input task segments from the output task, synchronizing the input task with the output task. If the output task deleted the segment, the input task creates a new segment and waits for input data. The entire process runs continuously, synchronized by mailboxes and segment availability. Additionally, the tasks can elect to wait for a specified amount of time at mailboxes, and if no segments appear, an error routine can alert the system operator that processing has stopped.

The mailbox method has several advantages over synchronization algorithms and polling tasks. The entire process is synchronized by the availability of data in segments, eliminating the need for algorithms and extra code; the same process applies whether the tasks operate at the same or different speeds. Also, burst input or output rates can be handled by adding buffers. For instance, if too much data arrives for the process or output tasks to handle immediately, the input task fills multiple buffers and passes them to mailbox A. The process task takes each segment in turn. After processing is completed, the segments are all sent to mailbox C, and the process waits for the next burst of data. The only interfaces between the tasks are mailboxes and segments, so tasks can be easily replaced or added to the processing loop; the same scheme works for larger or smaller segments.

Tasks and task scheduling

Tasks are independent bodies of executing code, initialized and scheduled by the kernel. Therefore, tasks must have iRMX 86 parameters like priority, initial memory resources, entry address, and other iRMX 86 data. A task is like an expanded subroutine managed by an operating system. The actual application code is written much the same as it is without an operating system except that requests are made using kernel calls.

Even though the system's multiple independent tasks appear to run simultaneously, only one task actually runs at one time. Some method of scheduling is needed to decide which task receives control of the system processor; this scheduling depends on the task priority. Since data coming into a system must not be missed, the input task has the highest priority. Data going out of the system are next in importance, so the output task has second priority; the sequential process task has the lowest priority. The scheduling algorithm is simple—the highest priority task that is ready to run will get control of the processor. This is an example of preemptive priority. In this case, ready to run means that a task is complete—it has a segment to fill and data coming in (input task), data to process (process task), or data to output (output task). For instance, if input data arrives when the process task is running and the input task has a buffer waiting for data, the input task will preempt the process task to receive

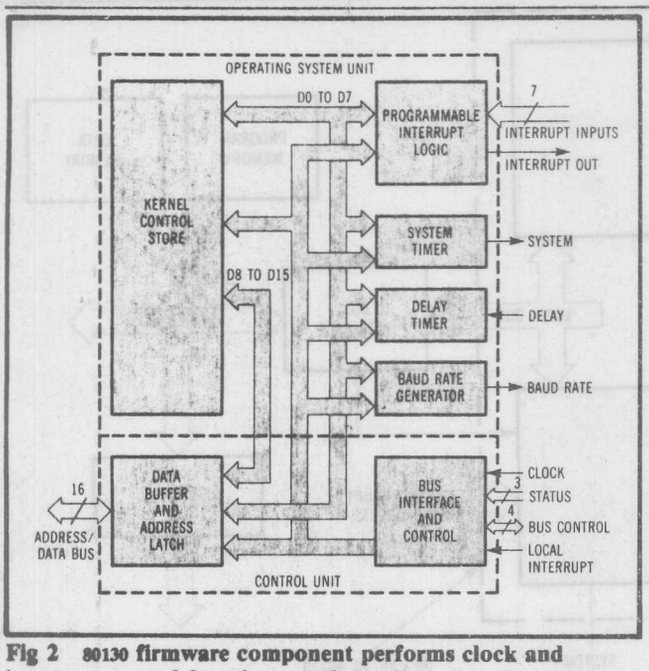


Fig 2 80130 firmware component performs clock and interrupt control functions, and supplies operating system primitives.

the data. If the input task is not running and the hardware driven by the output task is ready to output another data value, the output task will receive control of the processor.

Since the operating system schedules the tasks, each task is designed as though it has sole control of the processor. Tasks make system calls such as receive message, which may cause another task to run because no message is waiting. In addition, interrupts will likely cause a different task to run. The kernel can schedule the tasks because only interrupts or system calls can cause a higher priority task to become ready, and both of these are handled by the kernel. Thus any time an interrupt occurs or a system call is made, the kernel runs the highest priority task that is ready. The tasks are written without any code to manage scheduling. The kernel scheduling is general purpose, so adding new tasks to the system does not require modifying the scheduling functions. A system with work balanced among the tasks runs as though all tasks perform simultaneously.

The net result of task scheduling is that the system runs as fast as it can. When data come in, the input task will always get control of the processor. The output task will execute whenever it has data to send and the input task is not running. The process task will run whenever it has data and no other tasks are running. Also, tuning the system is easier with the standardized mailbox interfaces: slower tasks can be easily removed and replaced with faster tasks, and remaining tasks will not be affected.

In a multitasking system, multiple independent tasks execute concurrently. Buffer transfers occur through mailboxes rather than through a direct interface to tasks, and system functions not related to the primary data processing functions can be handled by other tasks. For example, a supervisory task that monitors a system console for operator requests can be added to the system at a lower priority than the process task. No changes to any scheduling algorithm would be required.

Interrupt management

The iRMX 86 kernel and the OSP provide two classes of interrupt management: interrupt handlers and interrupt tasks. An interrupt handler is a short procedure whose only function is to respond to the interrupt as quickly as possible. All interrupts become disabled in order to let the interrupt handler execute at top speed. Interrupt handlers can make only a few system calls. In the sample system, the interrupt procedure receives a data value, places it in a buffer, and returns. When the buffer is full, the interrupt handler notifies the interrupt task. Typical response time for an 8-MHz iAPX 86 processor, from the time an interrupt occurs until the interrupt handler gets control is 30 to 50 μs. In the unlikely event of a worst-case time, response time is about 160 μs.

Higher priority interrupts are enabled when an interrupt handler gets control, is 30 to 50 μs. In the unlikely task uses a mailbox to pass the full buffer on to the next task. Since both interrupts and tasks have priorities assigned to them, the kernel uses the task priority to

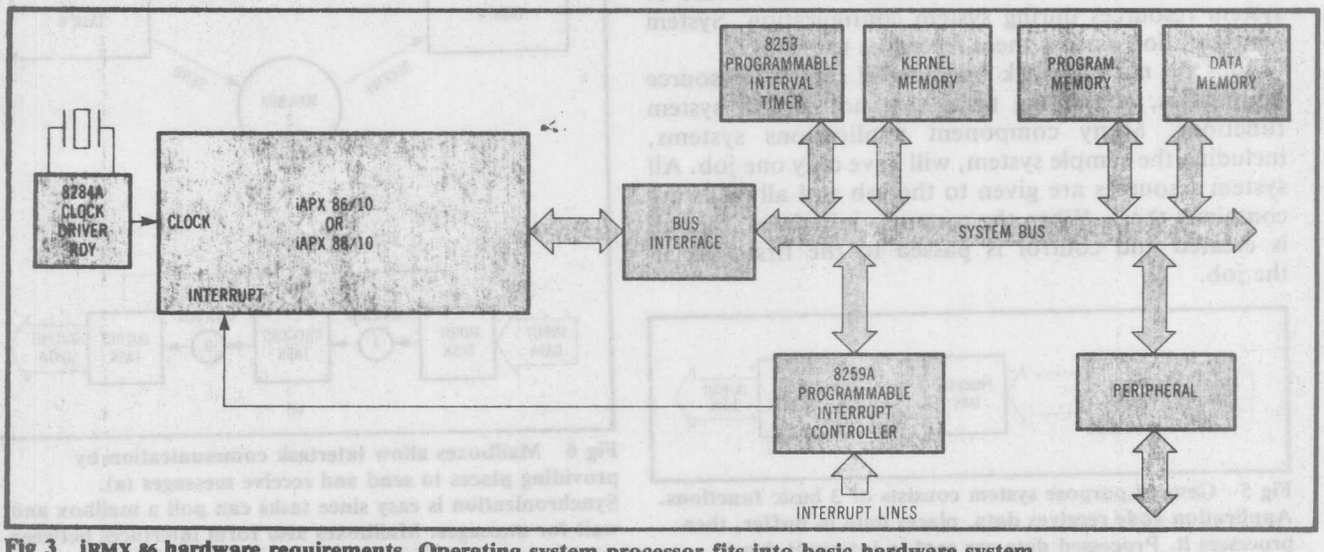


Fig 3 iRMX 86 hardware requirements. Operating system processor fits into basic hardware system for iRMX 86 and brings with it functions of kernel memory, 8259A programmable interrupt controller, and 8253 programmable interrupt timer.

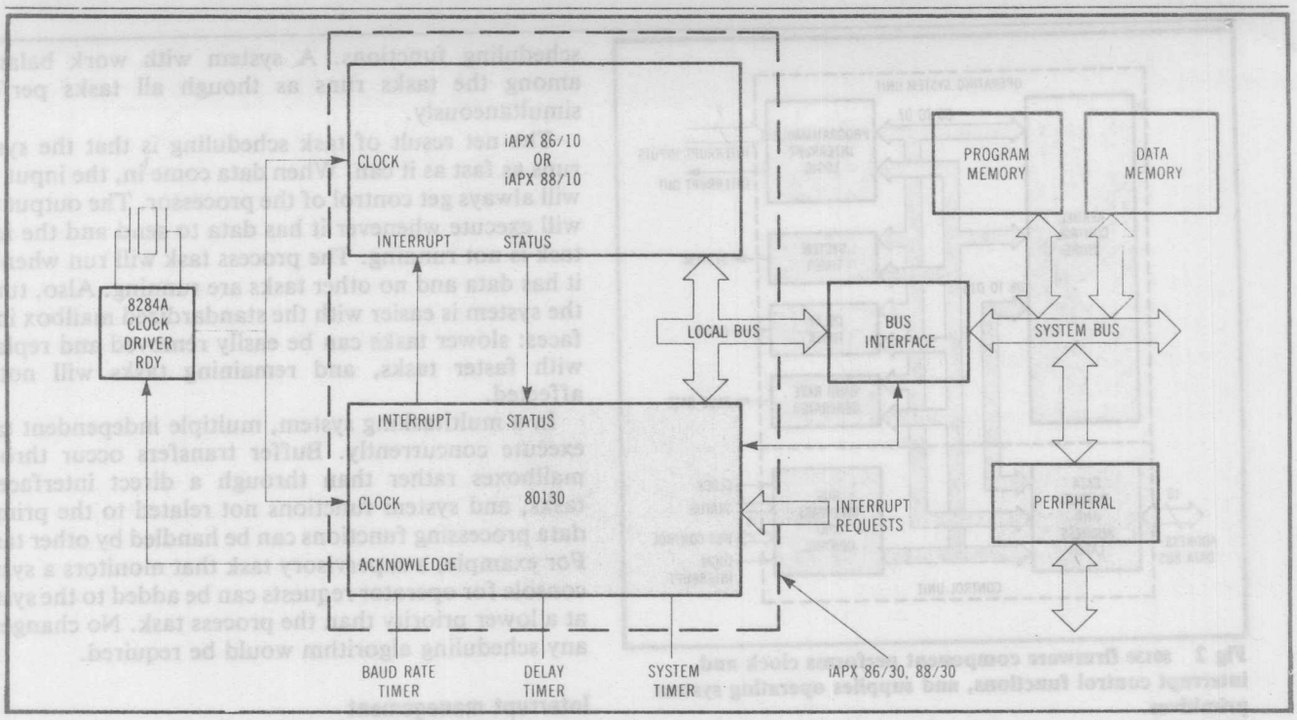


Fig 4 Basic hardware system with iAPX OSP. OSP replaces kernel code, programmable interrupt controller, and programmable interval timer.

determine if interrupts should be disabled or enabled. If the task priority is higher than an interrupt priority, that interrupt is disabled while the task is running. A priority level can be given to a task that disables all, some, or none of the interrupts: ie, defining a task that is more important than all interrupts (initialization task), more important than some interrupts (input task), or less important than all interrupts (processing task).

Multiprogramming

System parameters in a component system are normally well defined: RAM locations are fixed, code addresses are known, and address and I/O ports are specified. Application code usually depends on these parameters. If the system changes, substantial alterations are often needed in the application code. However, if an iRMX 86 operating system is used, the kernel is made aware of system resources during system configuration. System configuration assigns these resources to "jobs."

Jobs do not do work but instead serve as resource boundaries, containing tasks that accomplish system functions. Many component applications systems, including the sample system, will have only one job. All system resources are given to the job and all tasks are contained there. When the system is initialized, the job is created and control is passed to the first task in the job.

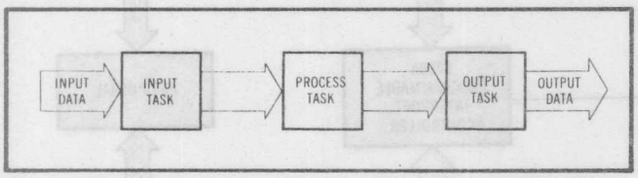


Fig 5 General purpose system consists of 3 basic functions. Application code receives data, places data in buffer, then processes it. Processed data are sent to interrupt driven output devices.

Multiprogramming occurs when a system has two or more jobs. The system boundaries provided by jobs confine errors and define limits for system resources such as memory. These boundaries limit the effect of one job on another. For instance, the system debugger is a separate job. During development, the sample processing system would look like Fig 7. After development, the debugger would be removed, leaving only the application system. The job environment of the processing system is not affected by adding or removing the debugger. The overall system will, of course, be affected

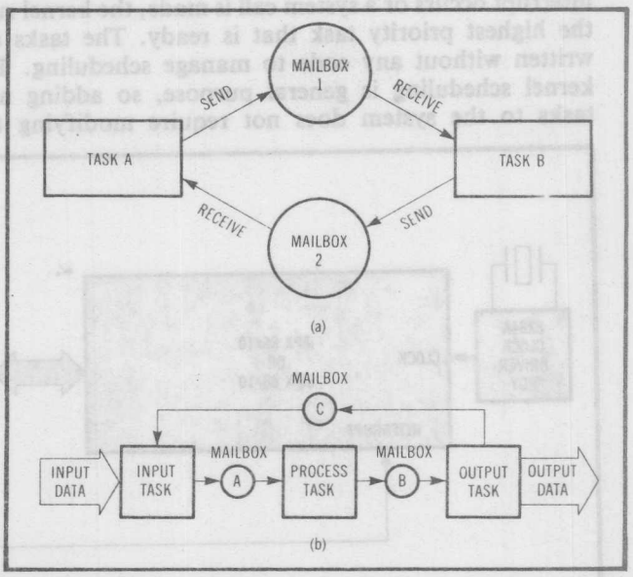


Fig 6 Mailboxes allow intertask communication by providing places to send and receive messages (a). Synchronization is easy since tasks can poll a mailbox and wait for messages. Mailboxes also form interfaces between tasks in application system (b) so tasks can be easily added or removed without changing code.

because removing the debugger will cause more system resources to be available for other jobs.

The jobs, tasks, segments, and mailboxes are part of a large set of system data types which are data structures managed by the operating system. System data types are manipulated only through system calls, which enforce the rules that govern their use. Together system data types and system calls form the application interface to the operating system. This interface provides not only a good boundary for error detection and debugging, but also common architecture that can be carried from application to application.

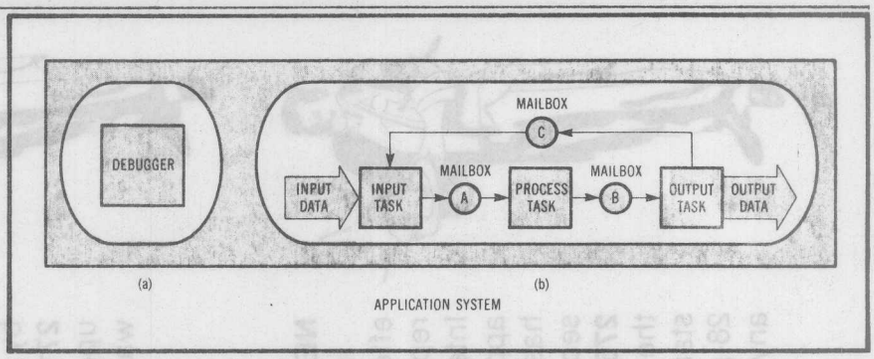


Fig 7 Job structure for development provides distinct boundaries so that a debugger (a) or other piece of development software can be used during system development and later removed without disturbing application job (b).

Debugging

The iRMX 86 operating system has a debugger that interprets and uses system data types, and manipulates them to control the system. For example, the processing flow in the sample system can be halted by the debugger when a segment is sent to mailbox A. Data flow through the system can be traced by halting or breakpointing the system as the segment goes from mailbox to mailbox.

Debugging is further aided by the modularity of the tasks and jobs. Modules limit error effects; the interfaces between the modules are well defined; and the modules are easily inserted or removed. A standard system debugger can be used for all applications, avoiding the need to develop specific diagnostic tools.

Conclusion

Multiprogramming and multitasking promote application code modularity, allowing applications to be created by adding new functions to old software. The same scheduling and kernel interfaces work for systems with only a few tasks, or systems with many tasks performing multiple processes. An entirely new process can be added to the example by adding more tasks. If the new and existing processes have nothing in common, the new process can be in a different job. If both processes can share general purpose tasks, such as output, the new process can be in the same job and use the mailbox interfaces to send data to one output task. If system designers are careful, they can design systems whose functions can be added in the field. Thus, expensive custom software will not have to be rewritten for each new application.

Users with a wide range of applications will find that this approach allows them to implement a corresponding range of capabilities, expanding an OSP based system up to a high level human interface. A complete iRMX 86 operating system includes extensive I/O capabilities, a debugger, an application loader, a bootstrap loader, and integrated user console functions. Such a system can perform general purpose processing and still provide all iRMX 86 facilities. With these features, one operating system can be used for current projects and expanded for future ones, minimizing software learning curves for new applications.

Bibliography

Introduction to the iRMX 86 Operating System, no 9803124, Intel Corp, Santa Clara, Calif, 1982

iRMX 86 Nucleus Reference Manual, no 9803122, Intel Corp, Santa Clara, Calif, 1981

Using the iRMX 86 Operating System on iAPX 86 Component Designs, Application Note AP110, Intel Corp, Santa Clara, Calif, 1981

J. Zarella, *Operating Systems Concepts and Principles,* Microcomputer Applications, Suisun City, Calif, 1979

Printed in U.S.A./BM-111/1182/10K/CD/CD
Order No. 210812-001

Create EPROM Magic!

*Intel's 28-pin **"Universal Site"** maximizes EPROM design and production flexibility.*

DESIGNING A SMALL SYSTEM?

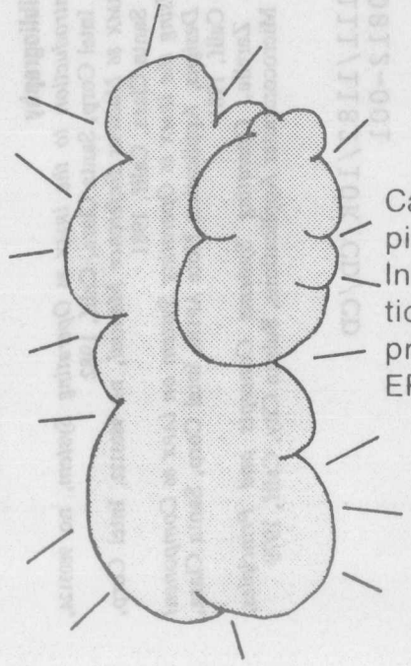
Intel's **2732A**/EPROMs fit nicely into the lower 24 pins of the "Universal Site." Use **2732As** today. Then add new features later by upgrading to the **2764** and **27128**. The "Universal Site" makes upgrades easier than waving a wand!

NEED VOLUME SUPPORT?

Intel's **2764** is today's most cost-effective EPROM. Call your Intel representative and we'll prove it! Intel's **2764** follows the JEDEC-approved 28-pin standard which has made it the most widely second-sourced 64K EPROM. Intel **2764s** are shipped to 0.1% AQL, the industry's tightest EPROM standard. And **2764s** plug into the 28-pin "Universal Site" without any slight-of-hand!

NEED BOARD SPACE?

Intel's **27128** is available NOW from your Intel distributor. Use it to "create" new board space without adding a board! For compact and portable designs it's the lowest-power per-bit EPROM available. HMOS*-E technology makes it affordable today! Get big performance from your 28-pin "Universal Site." Plug in the Intel **27128**, the 128K bit EPROM available **NOW!**



Make 24-pin sockets disappear! Catch all the magic of the Intel 28-pin "Universal Site!" Call your Intel representative for information on the "Universal Site." He'll provide the details . . . and the EPROMs that make it *work*!

*HMOS is a patented process of Intel Corporation.